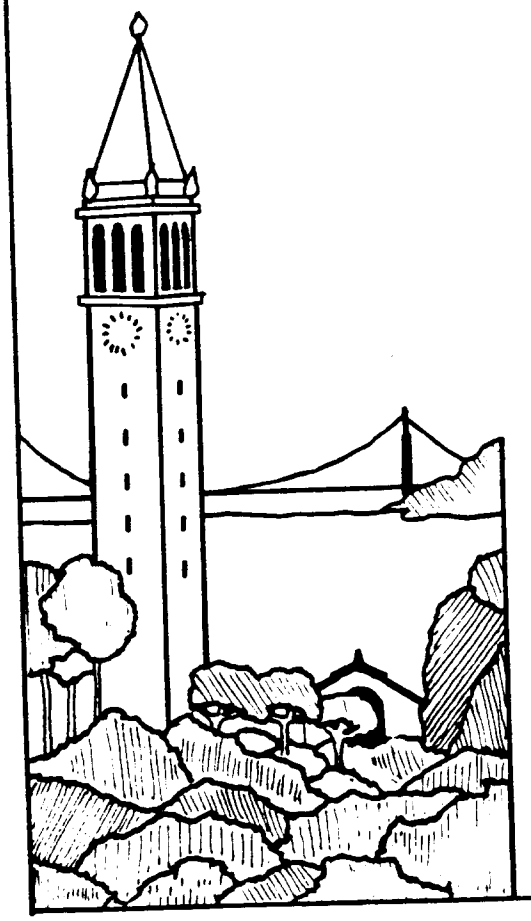
# Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System

*Brent Welch and John Ousterhout*



**Report No. UCB/CSD 86/261**

**October 1985**

**Computer Science Division (EECS)**
**University of California**
**Berkeley, California 94720**

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE | 2. REPORT TYPE | 3. DATES COVERED |
|---|---|---|
| **OCT 1985** | | **00-00-1985 to 00-00-1985** |

| 4. TITLE AND SUBTITLE | 5a. CONTRACT NUMBER |
|---|---|
| **Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System** | 5b. GRANT NUMBER |
| | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER |
| | 5e. TASK NUMBER |
| | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

**Prefix tables provide a mechanism for locating files in a system whose storage is distributed among many servers. The result is a single file system hierarchy visible uniformly and transparently to all clients. Each client of the filesystem maintains a local prefix table that identifies the server for a file based on the initial part of the file name. Prefix tables are built and modified using a simple broadcast protocol that is flexible enough to allow dynamic server reconfiguration and a simple form of replication.**

15. SUBJECT TERMS

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | |
| **unclassified** | **unclassified** | **unclassified** | **Same as Report (SAR)** | **16** | |

# Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System

*Brent Welch and John Ousterhout*

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

*ABSTRACT*

Prefix tables provide a mechanism for locating files in a system whose storage is distributed among many servers. The result is a single filesystem hierarchy visible uniformly and transparently to all clients. Each client of the filesystem maintains a local prefix table that identifies the server for a file based on the initial part of the file name. Prefix tables are built and modified using a simple broadcast protocol that is flexible enough to allow dynamic server reconfiguration and a simple form of replication. †

October 3, 1985

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and in part by the National Science Foundation under grant ECS-8351961.

# Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System

*Brent Welch and John Ousterhout*

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

## 1. Introduction

This paper describes a mechanism that uses prefix tables to locate files in a distributed filesystem. Prefix tables allow the storage for a filesystem to be distributed among many server machines in a network, yet provide client machines with the appearance of a single hierarchical filesystem. The distribution of storage is transparent to the user-level processes running on the clients. To them the filesystem appears just as it would if they were executing on a traditional timesharing system.

Each client of the filesystem maintains a small prefix table that identifies the server for a file based on the first few characters of the file's name. Prefix tables are constructed using a simple broadcast protocol and are updated automatically as the configuration of servers and files changes.

The prefix table mechanism has four attractive features. First, clients negotiate directly with file servers to build the prefix tables; this means there is no need for a separate name service. In Section 7.3 we suggest that if a name service is needed then it should be built on top of the filesystem, rather than vice versa. The second advantage of prefix tables is their dynamic nature: they permit the system to adapt gracefully as its configuration changes. The third advantage is that each client machine has its own prefix table; by placing slightly different entries in different clients, the mechanism can support private files and a simple form of replication. Finally, clients need not have any local disk storage: the information in the prefix tables is received over the network as part of file name lookup.

Section 2 provides background about the filesystem and the network environment. Section 3 outlines some approaches to server location other than prefix tables. Section 4 introduces prefix tables and their use in name lookup, and Section 5 describes how prefix tables are maintained using a simple broadcast protocol. Section 6 presents a few interesting applications of the prefix table mapping including private files and replication. Section 7 compares the prefix table mechanism to other naming services. Section 8 describes a prototype implementation of prefix tables that we have constructed as part of the UNIX operating system [7,4].

## 2. The Scenario

In designing the prefix table mechanism, our goal was to provide a distributed filesystem with the same appearance as the filesystems of typical timesharing systems like UNIX. This includes facilities such as hierarchical directories, symbolic links, working directories, and relative path names [7,4]. Besides the usual issues of storage and communication, which must be dealt with even in a centralized network filesystem, there are

two additional issues that must be resolved to distribute the filesystem: (a) how to divide the files among the various servers, and (b) how to locate the server for a file, given its name.

We assume that the division of files among servers is handled by system administrators much as the division of files among disk packs is handled today: the filesystem will be divided up into several subtrees (which we call *domains*) with each server providing storage for one or more of these domains. See Figure 1 for an example. Our goal is to support a user community of several hundred or at most a few thousand people. We expect that this requires a few tens of servers, each with a small number of domains. Measurements in [6] indicate that one local-area network, or at most a small number of connected networks, can provide enough bandwidth for a community of this size. We assume that a broadcast facility exists for this (inter)network.

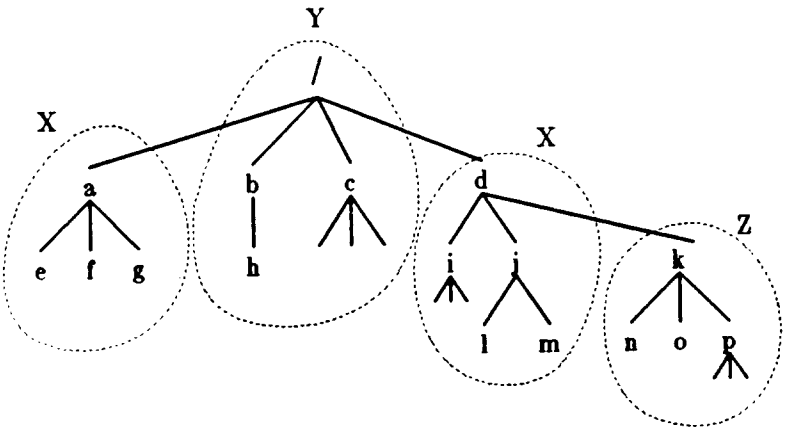



**Figure 1.** An example of a distributed filesystem. The hierarchy is divided up into four domains, each indicated by a dotted line. Server X contains storage for two of the domains, and servers Y and Z each handle a single domain.

The main topic of this paper is the second issue, how to locate a file given its name. This is handled by the *lookup* operation, which occurs whenever files are created or opened. In a traditional timesharing system, the lookup operation processes a hierarchical name such as **/a/b/c** by searching through directories, traversing symbolic links, and so on until the descriptor for the file is reached. The result of the lookup is a token that is used later on to provide quick access to the descriptor when the file is read or written. Typically, the token is an index into the operating system's internal table of open files.

In a distributed filesystem, the lookup operation invoked by clients must return two things: a server address and a token. The server address is used to send read and write requests to the appropriate server, and the token is passed to the server (as part of read and write requests) to identify the file being manipulated. In this case the token is typically an index into the server's table of open files; it saves the server from having to retranslate the file name on each read or write request.

## 3. Other Location Mechanisms

This section outlines three simple ways to locate servers. The prefix table mechanism includes aspects of each of these.

### 3.1. Broadcast

Perhaps the simplest way to locate files in a distributed system is to broadcast the file name to all servers during the lookup operation. Each server searches its local directories and the one successful server (if any) responds to the request with the server address and token for the file. The broadcast approach is simple to implement and adjusts automatically as the configuration of servers changes, but it suffers from high overhead: each server must process each file lookup. This lookup overhead would probably be the limiting factor in the size of the system.

### 3.2. Static Maps

A more efficient way of locating servers is to use part of the file name to identify the server. The simplest approach is for each file name to consist of a server name followed by a file name. For example, the name **X:/a/b** might refer to file **/a/b** on server **X**. In this scheme, each client keeps a small table that maps server names to server addresses. The table can be stored on the client's local disk or it might be loaded into memory as part of the bootstrapping process (for diskless clients). To lookup a file, the client consults the server map and sends a message directly to the server for the file. The server looks up the file and returns a token to use in read and write requests.

The advantage of the map approach is that it avoids the overhead of broadcasting on every file lookup. However, any change to the system configuration requires all the maps to be modified. For a system with hundreds or thousands of clients, this is a considerable overhead. A second disadvantage with static maps is that the server names appear in the file names. If a domain moves from one server to another then the names of all the files in that domain will change.

### 3.3. Name Servers

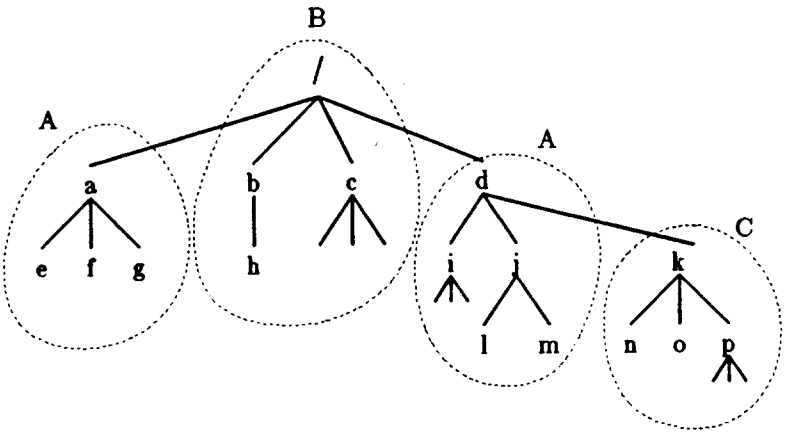
The static mapping approach can be extended by using a general-purpose name service like Grapevine [1] to map server names to addresses. Before opening a file a client first sends the file server name to the name service, which returns the address of the server for that file. The client then sends the name to the file server to complete the lookup process. Presumably the name service provides mechanisms for updating the map information; this avoids the inflexibility of the static map scheme. In the most general case, clients could send the entire file name to the name service, and the name servers could do the complete translation from file name to token.

There are two problems with the name server approach. The first is that it adds a third party to the lookup process: file names must be sent first to a name server and then to a file server. Any reliability or performance problems in the name service will have a direct impact on the filesystem. Second, the name server approach doesn't eliminate the update problem for the sever information; it simply transfers it from the filesystem to the name service. Implementing an efficient general-purpose name service with reliable distributed updates is a difficult programming task. It seemed both unnecessary and undesirable to us for a filesystem to depend on such a service.

## 4. Prefix Tables

Prefix tables combine the simplicity and efficiency of static maps with the flexibility of the broadcast approach. Each client keeps a server map called a *prefix table*, but instead of being static it is dynamically built and updated with a broadcast protocol. The server locations kept in the prefix table are hints that are corrected when they are found

to be wrong. This section describes how prefix tables are used during name lookups; Section 5 describes the mechanisms used to maintain the information in the tables.




| Prefix Table | | |
|---|---|---|
| Prefix | Server | Token |
| / | B | 17 |
| /a | A | 63 |
| /d | A | 44 |
| /d/k | C | 5 |

**Figure 2.** An example of a filesystem hierarchy with four domains handled by three different servers. The longest matching prefix in the prefix table determines which server is responsible for a particular file. For example, the file **/d/k/o** will be served by B, and the file **/a/e** will be served by A. When looking up the file **/d/k/o**, the client will send the name **o** to server C along with the token 5.

### 4.1. The Prefix Table as a Map

Each entry in a prefix table corresponds to one of the domains of the distributed filesystem: it contains the name of the topmost directory in the domain (called the *prefix* for the domain), the address of the server storing the domain, and a token. See Figure 2. To look up a file, the client searches the prefix table for directory names that match the first characters of the file name. If there is more than one matching prefix then the longest one is selected. The client strips the prefix from the file name and sends the remainder of the name to the selected server along with the token from the prefix table entry. The server uses the prefix token to locate the root directory of the domain, looks up the remainder of the file name, and replies with a token for the open file.

The tokens in prefix tables serve the same purpose as the tokens returned by the lookup operation: they allow the server to locate the descriptor for a file (in this case, the root directory for a domain) without having to repeat an expensive name translation. Prefix tokens are provided by servers to clients as part of the broadcast protocol described in Section 5.

The prefix table mechanism is more general than the static maps described in Section 3.2, since it allows the root of a domain to appear anywhere in the filesystem rather than just at the top level. Furthermore, the names of servers do not appear in file names. This makes the distribution transparent to users and permits domains to be moved from one server to another without affecting any file names.

In the examples so far, file names began at the root of the file hierarchy (they started with '/'). Most modern filesystems provide additional mechanisms for naming files, such as working directories and symbolic links. The remainder of this section describes how prefix tables can be used to implement these facilities for a distributed filesystem.

### 4.2. Working Directories and Relative File Names

Most filesystems allow each user to specify a *working directory*, and to name files relative to the working directory instead of the filesystem root. For example, in UNIX the file name **c** is relative (since it doesn't start with "/"). If the current working directory is **/a/b** then **c** names the file **/a/b/c**.

Timesharing systems like UNIX implement working directories by opening the directory and keeping its token as part of a process's state. When a program uses a relative file name the token for the working directory is used to locate the descriptor for the directory and the file name is looked up starting at that directory.

Working directories can be handled in almost exactly the same fashion in a distributed system. When a process specifies a new working directory, the prefix mechanism is used to open the working directory. Both the token and the server name are saved in the process's state. When a lookup operation detects a relative path name, it simply sends the name to the server for the current working directory along with the working directory token. From the server's standpoint the relative name lookup appears the same as the absolute name lookup described in the previous section: in either case the server receives a name to lookup and a token for the directory at which to begin the lookup.

### 4.3. Domain Crossings

There are several cases where the initial choice of server can be wrong. A file name can specify an arbitrary path through the file hierarchy and that path might cross the boundary between server domains. When this happens the initial server detects that the file name is leaving its domain and it returns a new file name to the client. The client uses the new name to select a new entry in its prefix table, and initiates a new lookup operation with that server. Several such retries may be necessary before the correct server for a file is finally found. The next subsections discuss specific situations where domain crossings occur.

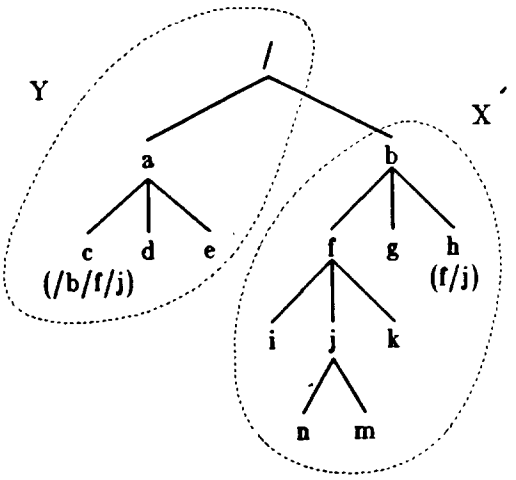



**Figure 3.** An example of symbolic link usage. The symbolic links at **/a/c** and **/b/h** both refer to the same directory. The link at **/a/c** is absolute: when encountered, the name is returned to the client for prefix processing. The link at **/b/h** is relative so server X can continue processing it. For example, when looking up the name **/b/h/m**, the client will pass **h/m** to server X along with a token for **b**. X will encounter the symbolic link at **h**, and will combine that with the remainder of the name to form a new name **f/j/m**. Since this is a relative name, it will keep processing it from **b**, eventually reaching file **m**.

### 4.3.1. Symbolic Links

A symbolic link is just a file whose contents are another file name [4]. When a server encounters a symbolic link during name lookup it prepends the contents of the symbolic link to the remaining portion of the file name before continuing the lookup. If this results in a relative path name then the server continues the lookup from the directory that contained the symbolic link. If, however, the new name is an absolute name (e.g. it starts with "/" in UNIX), then the server sends the new name back to the client. The client looks up this new name in its prefix table and retries the lookup with a new server. See Figure 3 for examples.

### 4.3.2. Parent Directories

Another common feature in filesystems is a notion for referring to the parent of a directory. For example, in UNIX the name ".." refers to the parent of the current directory. If a server encounters a sequence of ".." components in a file name, it may ascend the filesystem hierarchy past the root of its domain. At this point the server can no longer continue processing the name, so it returns the remainder of the name to the client. The client combines the remainder of the name with the prefix for the domain that was just exited to form a new absolute name. It then searches the prefix table for this name and chooses a new server to process it.

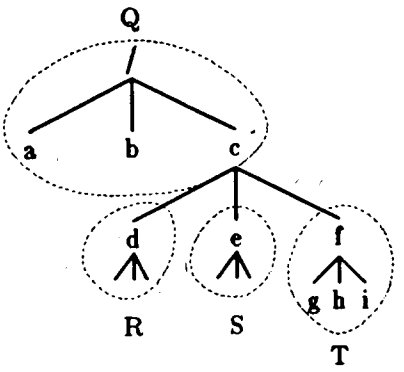



**Figure 4.** This filesystem is divided into four domains stored by servers Q, R, S, and T. If a user's working directory is **/c** then the relative file name **f/h** will cross a domain boundary. The name is initially sent to server Q. When Q encounters the remote link at **/c/f** it returns the name **/c/f/h** back to the client for prefix processing. The client will then select server T and retry the lookup.

### 4.3.3. Remote Links

A file name can descend down into a domain as well as ascend off the top as described in the last sub-section. This happens when the root of a domain is beneath the working directory in the filesystem hierarchy. See Figure 4 for an example. The solution to this problem is to place a marker in the filesystem that indicates the start of a new domain. The marker is just a special kind of file called a *remote link*. A remote link is a kind of circular symbolic link; its contents are its own complete name. When a server encounters a remote link it does the same kind of expansion as with a symbolic link and returns the new file name to the client for prefix matching.

For each domain in the distributed filesystem there is a remote link in the parent directory of the domain's root. In addition to trapping relative names that cross domains, the remote links give the domains a tangible appearance in directory listings. For example, in Figure 4 a directory listing for **/c** will show the entries **d**, **e**, and **f**, all of which are remote links. Remote links also assist in building up prefix tables dynamically (see Section 5 below).

## 5. Managing Prefix Tables

As explained so far, prefix tables appear much like the static maps of Section 3.2. There is a substantial difference, however. Prefix tables are not static: they are created and modified dynamically using a broadcast protocol. This allows the tables to be filled in incrementally and to adapt as the server configuration changes.

A single broadcast operation is used to maintain prefix tables. To obtain prefix information, a client broadcasts a file name. One of the servers responds with the prefix table entry for that file, including the string to use as prefix, the server's address, and the token corresponding to the domain. This operation is very similar to the procedure used for broadcast lookup in Section 3.1, except that it is only used to fill in and update prefix tables.

Initially, each client starts with an empty prefix table. When a lookup operation finds no matching prefix in its table then it broadcasts the file name as described above. It uses the response to create a new entry in the prefix table. If there is no response to the broadcast then the file is not accessible and an error is returned to the user. Entries

are added to the prefix table only when needed: a domain that has never been accessed will not appear in the prefix table.

Remote links play an important role in building up prefix tables by indicating when new prefixes are required. For example, consider a filesystem containing two domains, / and /a. If a client's prefix table contains only the entry / and it attempts to open the file /a/b it will send the request to the server for / instead of /a. The server for / encounters the remote link for /a and returns information about it to the client so that the client can broadcast for a new prefix. Without the remote link it would appear that the file /a/b did not exist.

If a server crashes, clients will get timeout errors when they attempt to invoke operations on that server. When this happens, the client returns an error to the user process (just as if a disk had suddenly gone off-line). In addition, the client removes its prefix table entries for the crashed server. Until the server reboots, clients will be unable to access its files: each attempted access will result in a broadcast for a prefix, followed by a timeout. However, when the server reboots then its files will be accessible again: the next attempted access will result in a successful broadcast that will re-establish the prefix table entry. This same mechanism also works if the server reboots at a different network address, or if the domains are moved to other servers. It isn't necessary to reboot clients after server crashes, although programs running at the time of a crash will probably have to be restarted if they were using files on the crashed server.

## 6. Private Storage and Replication

Until now we have been considering the case where all the clients have the same view of the filesystem. We expect this to be the most common case. However, each client's view is defined by its own private prefix table, and the prefix tables could vary from client to client. Different clients may have different sets of prefixes or may use different servers for the same prefix. They may even have more than one prefix that refers to the same domain.

This flexibility is possible because the information in a prefix table entry is reached by negotiation between the client and a server rather than being hardwired into the filesystem. Either the client or the server may take actions that result in different prefix table entries in different clients. Three examples are described below; private files, replication of heavily-used files, and support for heterogeneous clients.

### 6.1. Private Files

A client may wish to keep private files on a local disk for performance or security reasons. It can accomplish this by acting as a server for the local storage and placing an entry for the domain in its prefix table. The workstation can guarantee the privacy of the files by refusing to answer broadcast queries about them. One use of this facility in UNIX-like systems might be for the directory **/usr/tmp**, which holds temporary files generated by many UNIX programs. Every workstation needs access to **/usr/tmp**, but workstations with local disks would probably prefer to use their own disks for the temporary space. They can set up their own **/usr/tmp** domains for private use, with a network file server providing a public version of **/usr/tmp** for diskless clients. All broadcasts for the **/usr/tmp** prefix would be handled by the public server. Workstations with local disks wouldn't broadcast for the **/usr/tmp** prefix, but would initialize their prefix tables with a prefex referring to the local storage.

### 6.2. Replication

In the private-storage example, a particular client conspired to arrange for special entries in its prefix table. Servers may also conspire to give different clients different prefix table entries: this provides for a simple form of file replication. For example, in a large network there may be a lot of traffic to certain files such as the binary images of system programs. The domain containing these files could be replicated on two or more different servers. The servers can then negotiate between themselves to decide which server will service which clients. When a client broadcasts for information about the prefix, only one of the servers responds. If a server becomes overloaded it can arrange with another server to handle particular clients, then ignore future open requests from those clients (while continuing to service read and write requests for already-open files). The clients will rebroadcast the prefix and receive information from the new server. Shuffling between servers in this way is completely transparent to processes running on the clients.

In order to update replicated files, clients must be able to access each of the copies of the file. The prefix tables can provide this facility, although they do not provide any assistance in making atomic updates to all the copies. Access to the different copies of a domain can be accomplished with extra prefixes that reference particular copies explicitly. See Figure 5 for an example.

| Client 1 Prefix Table | | | Client 2 Prefix Table | | |
|---|---|---|---|---|---|
| Prefix | Server | Index | Prefix | Server | Index |
| / | A | 3 | / | A | 3 |
| /bin | B | 13 | /bin | C | 25 |
| /bin1 | B | 13 | /bin1 | B | 13 |
| /bin2 | C | 25 | /bin2 | C | 25 |

**Figure 5.** An example of replication. Identical copies of domain **bin** are stored on servers B and C. The prefix tables cause Client 1's accesses to **/bin** to be handled by server B, while Client 2's accesses are handled by server C. If client 1 needs to access the copy of **/bin** on server C (e.g. to write a new version of one of the files), it can do so using prefix **/bin2**.

This same technique for replication can be used to allow clients with different instruction sets to share the filesystem, even though they require different binary files for system programs. This is handled with several different domains, each containing versions of system programs for a particular instruction set. A single prefix, say **/bin**, refers to all of these domains. Servers can consult tables stored in the filesystem to determine which clients have which instruction sets and respond appropriately to prefix broadcasts so that each client uses the version of **/bin** corresponding to its instruction set.

## 7. Comparisons

### 7.1. Prefixes vs. Network Filesystems

Most of the network filesystems discussed in the literature to date have been concerned primarily with issues of data storage and synchronization, rather than naming. The earliest filesystems, such as WFS [8], XDFS[5], and CFS [5], did not even provide directories; files were named by unique identifiers (similar to the tokens we've used here), and more user-friendly naming facilities were to be provided in an unspecified fashion by client programs. More recent systems, discussed below, do provide naming facilities, but

mostly along the lines of the simple mechanisms presented in Section 3.

Apollo's Aegis operating system [3] provides what is probably the most widely-used network filesystem. It includes a network-wide name space, but files on remote machines must be identified by their machine name as in Section 3.2; the mapping of files to servers is not transparent as it is with prefix tables.

LOCUS is a distributed UNIX system that provides a transparently-distributed filesystem [9]. Here too, though, the focus has been on storage, replication, and transactions, rather than naming. Although file names do not contain machine names, servers are located with static mount tables replicated at each site. The LOCUS authors note that this scheme would have to be modified in order for the system to scale up to hundreds or thousands of sites. LOCUS also uses a different lookup mechanism in which clients process the directories themselves by reading the directory files over the network. This results in more network traffic than a scheme where directory lookups are handled by the servers.

The closest existing mechanism to our prefix scheme is a new naming mechanism recently added to the V system [2]. It combines naming facilities with object storage and uses prefixes for name storage in a way very similar to what is described here.

### 7.2. Prefixes vs. Timesharing

Prefix tables present a view of the filesystem identical to that seen by a user process in a uniprocessor timesharing system, with one exception. The prefix tables bypass part of the directory lookup mechanism, and this can alter the permission checking done during lookup. For example, if there is a prefix **/a/b/c** and a client looks up the file **/a/b/c/d**, neither of the directories **/a** or **/a/b** is examined: the client communicates directly with the server containing **/a/b/c**. This means that any access controls in **/a** or **/a/b** will be ignored. The effect is that all programs implicitly have search permission along the paths to all the prefixes. If access to a prefix is to be restricted, it must be restricted with access controls at the level of the prefix or below. The directory bypass has advantages in performance and reliability, though: the root server will be less congested and clients can still access files even when the root server is unavailable.

### 7.3. Name Service vs. Filesystem

Much of the distributed-system research to date appears to be based on the assumption that a general-purpose name service lies at the heart of any distributed system, and that other facilities such as filesystems should be built on top of the name service. We disagree with this approach. On the contrary, we believe that a general-purpose filesystem lies at the heart of distributed systems just as it does in the timesharing world. It is unclear whether name servers are even necessary given a filesystem, since the directory structure of the filesystem already provides a naming service. Instead of attaching information to well-known names in a name server, it can be left in well-known files in the filesystem.

Although a filesystem requires some sort of simple name service for locating file servers, a special-purpose mechanism like prefixes is much simpler and more efficient than a general-purpose name service like that provided by Grapevine [1]. On the other hand, a powerful name service like Grapevine requires data storage facilities. Without a pre-existing filesystem, the name server must duplicate these facilities. If a name service is necessary, it seems to us that the best way to construct it is to use the filesystem facilities as a base. In this fashion the name service can focus on issues of redundancy, update, etc. without having to worry about low-level storage mechanisms.

## 8. A Prototype Implementation

We have implemented a prototype version of the prefix mechanism, running on a cluster of about ten Sun workstations as an extension to the UNIX operating system. The implemented mechanism was simplified slightly in order to avoid having to add remote links to UNIX. Instead of remote links, the implementation uses symbolic links. These work just as well as remote links except that servers and clients cannot tell whether or not a symbolic link indicates the root of a new domain; this leads to problems in building up prefix tables dynamically. As a consequence, all prefix names are entered into each prefix table statically at boot time so that all domains are identified. The server and token information for prefix table entries is still located using broadcasts, but the names of the prefixes are static.

## 9. Conclusion

Prefix tables are a special-purpose mechanism used to locate file servers. The single shared file hierachy provided by the servers replaces a general name service as the foundation of a distributed system. A client's view of the filesystem is kept consistent with changes and additions to the filesystem by using a broadcast protocol to update the prefix tables. Furthermore, by adjusting different clients' prefix tables the mechanism supports private files and file replication.

## 10. Acknowledgements

Garth Gibson and Susan Eggers developed the basic prefix table idea during a class project last spring, and Walter Scott participated in the initial design of the prototype implementation.

## 11. References

[1] Andrew D. Birrell et al. "Grapevine: An Exercise in Distributed Computing." *Communications of the ACM*, Vol. 25, No. 4, April 1982, pp. 260-274.

[2] David R. Cheriton and Timothy P. Mann. "A Decentralized Naming Architecture." Internal Report, Computer Science Department, Stanford University, July 1985.

[3] Paul J. Leach et. al. "The Architecture of an Integrated Local Network." *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, No. 5, November 1983, pp. 842-857.

[4] Marshall K. McKusick et al. "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[5] James G. Mitchell and Jeremy Dion. "A Comparison of Two Network-Based File Servers." *Communications of the ACM*, Vol. 25, No. 4, April 1982, pp. 233-245.

[6] John K. Ousterhout et al. "A Trace-Driven Analysis of the 4.2 BSD UNIX File System." *Proceedings of the 10th Symposium on Operating Systems Principles*, to appear, December 1985.

[7] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-Sharing System." *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

[8] Daniel Swinehart, Gene McDaniel and David Boggs. "WFS: A Simple Shared File System for a Distributed Environment." *Proceedings of the 7th Symposium on Operating Systems Principles*, December, 1979, pp. 9-17.

[9] Bruce Walker et. al., "The LOCUS Distributed Operating System." Proceedings of the 9th SOSP, *Operating Systems Review*, Vol. 17, No. 5, November 1983, pp. 49-

70.